

# Study guide: Finite difference methods for wave motion

Hans Petter Langtangen<sup>1,2</sup>

<sup>1</sup>Center for Biomedical Computing, Simula Research Laboratory

<sup>2</sup>Department of Informatics, University of Oslo

Sep 10, 2015

## Finite difference methods for waves on a string

Waves on a string can be modeled by the *wave equation*

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

$u(x, t)$  is the displacement of the string

[Demo of waves on a string.](#)

## The complete initial-boundary value problem

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \quad x \in (0, L), \quad t \in (0, T] \quad (1)$$

$$u(x, 0) = I(x), \quad x \in [0, L] \quad (2)$$

$$\frac{\partial}{\partial t} u(x, 0) = 0, \quad x \in [0, L] \quad (3)$$

$$u(0, t) = 0, \quad t \in (0, T] \quad (4)$$

$$u(L, t) = 0, \quad t \in (0, T] \quad (5)$$

## Input data in the problem

- Initial condition  $u(x, 0) = I(x)$ : initial string shape
- Initial condition  $u_t(x, 0) = 0$ : string starts from rest
- $c = \sqrt{T/\varrho}$ : velocity of waves on the string
- ( $T$  is the tension in the string,  $\varrho$  is density of the string)

- Two boundary conditions on  $u$ :  $u = 0$  means fixed ends (no displacement)

Rule for number of initial and boundary conditions:

- $u_{tt}$  in the PDE: two initial conditions, on  $u$  and  $u_t$
- $u_t$  (and no  $u_{tt}$ ) in the PDE: one initial conditions, on  $u$
- $u_{xx}$  in the PDE: one boundary condition on  $u$  at each boundary point

### Demo of a vibrating string ( $C = 0.8$ )

- Our numerical method is sometimes exact (!)
- Our numerical method is sometimes subject to serious non-physical effects

### Demo of a vibrating string ( $C = 1.0012$ )

Ooops!

### Step 1: Discretizing the domain

Mesh in time:

$$0 = t_0 < t_1 < t_2 < \cdots < t_{N_t-1} < t_{N_t} = T \quad (6)$$

Mesh in space:

$$0 = x_0 < x_1 < x_2 < \cdots < x_{N_x-1} < x_{N_x} = L \quad (7)$$

Uniform mesh with constant mesh spacings  $\Delta t$  and  $\Delta x$ :

$$x_i = i\Delta x, \quad i = 0, \dots, N_x, \quad t_i = n\Delta t, \quad n = 0, \dots, N_t \quad (8)$$

### The discrete solution

- The numerical solution is a mesh function:  $u_i^n \approx u_e(x_i, t_n)$
- Finite difference stencil (or scheme): equation for  $u_i^n$  involving neighboring space-time points



### Step 2: Fulfilling the equation at the mesh points

Let the PDE be satisfied at all *interior* mesh points:

$$\frac{\partial^2}{\partial t^2} u(x_i, t_n) = c^2 \frac{\partial^2}{\partial x^2} u(x_i, t_n), \quad (9)$$

for  $i = 1, \dots, N_x - 1$  and  $n = 1, \dots, N_t - 1$ .

For  $n = 0$  we have the initial conditions  $u = I(x)$  and  $u_t = 0$ , and at the boundaries  $i = 0, N_x$  we have the boundary condition  $u = 0$ .

### Step 3: Replacing derivatives by finite differences

Widely used finite difference formula for the second-order derivative:

$$\frac{\partial^2}{\partial t^2} u(x_i, t_n) \approx \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = [D_t D_t u]_i^n$$

and

$$\frac{\partial^2}{\partial x^2} u(x_i, t_n) \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} = [D_x D_x u]_i^n$$

### Step 3: Algebraic version of the PDE

Replace derivatives by differences:

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = c^2 \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}, \quad (10)$$

In operator notation:

$$[D_t D_t u = c^2 D_x D_x]_i^n \quad (11)$$

### Step 3: Algebraic version of the initial conditions

- Need to replace the derivative in the initial condition  $u_t(x, 0) = 0$  by a finite difference approximation
- The differences for  $u_{tt}$  and  $u_{xx}$  have second-order accuracy
- Use a centered difference for  $u_t(x, 0)$

$$[D_{2t} u]_i^n = 0, \quad n = 0 \quad \Rightarrow \quad u_i^{n-1} = u_i^{n+1}, \quad i = 0, \dots, N_x$$

The other initial condition  $u(x, 0) = I(x)$  can be computed by

$$u_i^0 = I(x_i), \quad i = 0, \dots, N_x$$

### Step 4: Formulating a recursive algorithm

- Nature of the algorithm: compute  $u$  in space at  $t = \Delta t, 2\Delta t, 3\Delta t, \dots$
- Three time levels are involved in the general discrete equation:  $n + 1$ ,  $n$ ,  $n - 1$
- $u_i^n$  and  $u_i^{n-1}$  are then already computed for  $i = 0, \dots, N_x$ , and  $u_i^{n+1}$  is the unknown quantity

Write out  $[D_t D_t u = c^2 D_x D_x]_i^n$  and solve for  $u_i^{n+1}$ ,

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2 (u_{i+1}^n - 2u_i^n + u_{i-1}^n) \quad (12)$$

### The Courant number

$$C = c \frac{\Delta t}{\Delta x}, \quad (13)$$

is known as the (dimensionless) *Courant number*

**Observe.** There is only one parameter,  $C$ , in the discrete model:  $C$  lumps mesh parameters  $\Delta t$  and  $\Delta x$  with the only physical parameter, the wave velocity  $c$ . The value  $C$  and the smoothness of  $I(x)$  govern the quality of the numerical solution.

## The finite difference stencil



## The stencil for the first time level

- Problem: the stencil for  $n = 1$  involves  $u_i^{-1}$ , but time  $t = -\Delta t$  is outside the mesh
- Remedy: use the initial condition  $u_t = 0$  together with the stencil to eliminate  $u_i^{-1}$

Initial condition:

$$[D_{2t}u = 0]_i^0 \Rightarrow u_i^{-1} = u_i^1$$

Insert in stencil  $[D_t D_t u = c^2 D_x D_x]_i^0$  to get

$$u_i^1 = u_i^0 - \frac{1}{2}C^2 (u_{i+1}^n - 2u_i^n + u_{i-1}^n) \quad (14)$$

## The algorithm

1. Compute  $u_i^0 = I(x_i)$  for  $i = 0, \dots, N_x$
2. Compute  $u_i^1$  by (14) and set  $u_i^1 = 0$  for the boundary points  $i = 0$  and  $i = N_x$ , for  $n = 1, 2, \dots, N - 1$ ,
3. For each time level  $n = 1, 2, \dots, N_t - 1$ 
  - (a) apply (12) to find  $u_i^{n+1}$  for  $i = 1, \dots, N_x - 1$
  - (b) set  $u_i^{n+1} = 0$  for the boundary points  $i = 0, i = N_x$ .

## Moving finite difference stencil

[web page](#) or a [movie file](#).

## Sketch of an implementation (1)

- Arrays:
  - `u[i]` stores  $u_i^{n+1}$
  - `u_1[i]` stores  $u_i^n$
  - `u_2[i]` stores  $u_i^{n-1}$

**Naming convention.** `u` is the unknown to be computed (a spatial mesh function), `u_k` is the computed spatial mesh function `k` time steps back in time.

## PDE solvers should save memory

**Important to minimize the memory usage.** The algorithm only needs to access the *three most recent time levels*, so we need only three arrays for  $u_i^{n+1}$ ,  $u_i^n$ , and  $u_i^{n-1}$ ,  $i = 0, \dots, N_x$ . Storing all the solutions in a two-dimensional array of size  $(N_x + 1) \times (N_t + 1)$  would be possible in this simple one-dimensional PDE problem, but not in large 2D problems and not even in small 3D problems.

## Sketch of an implementation (2)

```
# Given mesh points as arrays x and t (x[i], t[n])
dx = x[1] - x[0]
dt = t[1] - t[0]
C = c*dt/dx          # Courant number
Nt = len(t)-1
C2 = C**2             # Help variable in the scheme

# Set initial condition u(x,0) = I(x)
for i in range(0, Nx+1):
    u_1[i] = I(x[i])

# Apply special formula for first step, incorporating du/dt=0
for i in range(1, Nx):
    u[i] = u_1[i] - 0.5*C**2*(u_1[i+1] - 2*u_1[i] + u_1[i-1])
u[0] = 0; u[Nx] = 0    # Enforce boundary conditions

# Switch variables before next step
u_2[:,], u_1[:,] = u_1, u

for n in range(1, Nt):
    # Update all inner mesh points at time t[n+1]
    for i in range(1, Nx):
        u[i] = 2*u_1[i] - u_2[i] - \
            C**2*(u_1[i+1] - 2*u_1[i] + u_1[i-1])
```

```

# Insert boundary conditions
u[0] = 0; u[Nx] = 0

# Switch variables before next step
u_2[:, u_1[:, u_1, u

```

## Verification

- Think about testing and verification before you start implementing the algorithm!
- Powerful testing tool: method of manufactured solutions and computation of convergence rates
- Will need a source term in the PDE and  $u_t(x, 0) \neq 0$
- Even more powerful method: exact solution of the scheme

## A slightly generalized model problem

Add source term  $f$  and nonzero initial condition  $u_t(x, 0)$ :

$$u_{tt} = c^2 u_{xx} + f(x, t), \quad (15)$$

$$u(x, 0) = I(x), \quad x \in [0, L] \quad (16)$$

$$u_t(x, 0) = V(x), \quad x \in [0, L] \quad (17)$$

$$u(0, t) = 0, \quad t > 0, \quad (18)$$

$$u(L, t) = 0, \quad t > 0 \quad (19)$$

## Discrete model for the generalized model problem

$$[D_t D_t u = c^2 D_x D_x + f]_i^n \quad (20)$$

Writing out and solving for the unknown  $u_i^{n+1}$ :

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2(u_{i+1}^n - 2u_i^n + u_{i-1}^n) + \Delta t^2 f_i^n \quad (21)$$

## Modified equation for the first time level

Centered difference for  $u_t(x, 0) = V(x)$ :

$$[D_{2t} u = V]_i^0 \Rightarrow u_i^{-1} = u_i^1 - 2\Delta t V_i,$$

Inserting this in the stencil (21) for  $n = 0$  leads to

$$u_i^1 = u_i^0 - \Delta t V_i + \frac{1}{2} C^2 (u_{i+1}^0 - 2u_i^0 + u_{i-1}^0) + \frac{1}{2} \Delta t^2 f_i^0 \quad (22)$$

## Using an analytical solution of physical significance

- Standing waves occur in real life on a string
- Can be analyzed mathematically (known exact solution)

$$u_e(x, y, t) = A \sin\left(\frac{\pi}{L}x\right) \cos\left(\frac{\pi}{L}ct\right) \quad (23)$$

- PDE data:  $f = 0$ , boundary conditions  $u_e(0, t) = u_e(L, 0) = 0$ , initial conditions  $I(x) = A \sin\left(\frac{\pi}{L}x\right)$  and  $V = 0$
- Note:  $u_i^{n+1} \neq u_e(x_i, t_{n+1})$ , and we do not know the error, so testing must aim at reproducing the expected convergence rates

## Manufactured solution: principles

- Disadvantage with the previous physical solution: it does not test  $V \neq 0$  and  $f \neq 0$
- Method of manufactured solution:
  - Choose some  $u_e(x, t)$
  - Insert in PDE and fit  $f$
  - Set boundary and initial conditions compatible with the chosen  $u_e(x, t)$

## Manufactured solution: example

$$u_e(x, t) = x(L - x) \sin t$$

PDE  $u_{tt} = c^2 u_{xx} + f$ :

$$-x(L - x) \sin t = -2 \sin t + f \quad \Rightarrow \quad f = (2 - x(L - x)) \sin t$$

Implied initial conditions:

$$\begin{aligned} u(x, 0) &= I(x) = 0 \\ u_t(x, 0) &= V(x) = -x(L - x) \end{aligned}$$

Boundary conditions:

$$u(x, 0) = u(x, L) = 0$$



## Testing a manufactured solution

- Introduce common mesh parameter:  $h = \Delta t$ ,  $\Delta x = ch/C$
- This  $h$  keeps  $C$  and  $\Delta t/\Delta x$  constant
- Select coarse mesh  $h$ :  $h_0$
- Run experiments with  $h_i = 2^{-i}h_0$  (halving the cell size),  $i = 0, \dots, m$
- Record the error  $E_i$  and  $h_i$  in each experiment
- Compute pairwise convergence rates  $r_i = \ln E_{i+1}/E_i / \ln h_{i+1}/h_i$
- Verification:  $r_i \rightarrow 2$  as  $i$  increases

## Constructing an exact solution of the discrete equations

- Manufactured solution with computation of convergence rates: much manual work
- Simpler and more powerful: use an exact solution for  $u_i^n$
- A linear or quadratic  $u_e$  in  $x$  and  $t$  is often a good candidate

## Analytical work with the PDE problem

Here, choose  $u_e$  such that  $u_e(x, 0) = u_e(L, 0) = 0$ :

$$u_e(x, t) = x(L - x)(1 + \frac{1}{2}t),$$

Insert in the PDE and find  $f$ :

$$f(x, t) = 2(1 + t)c^2$$

Initial conditions:

$$I(x) = x(L - x), \quad V(x) = \frac{1}{2}x(L - x)$$

## Analytical work with the discrete equations (1)

We want to show that  $u_e$  also solves the discrete equations!

Useful preliminary result:

$$[D_t D_t t^2]^n = \frac{t_{n+1}^2 - 2t_n^2 + t_{n-1}^2}{\Delta t^2} = (n+1)^2 - n^2 + (n-1)^2 = 2 \quad (24)$$

$$[D_t D_t t]^n = \frac{t_{n+1} - 2t_n + t_{n-1}}{\Delta t^2} = \frac{((n+1) - n + (n-1))\Delta t}{\Delta t^2} = 0 \quad (25)$$

Hence,

$$[D_t D_t u_e]_i^n = x_i(L - x_i)[D_t D_t(1 + \frac{1}{2}t)]^n = x_i(L - x_i)\frac{1}{2}[D_t D_t t]^n = 0$$

### Analytical work with the discrete equations (1)

$$\begin{aligned} [D_x D_x u_e]_i^n &= (1 + \frac{1}{2}t_n)[D_x D_x(xL - x^2)]_i = (1 + \frac{1}{2}t_n)[LD_x D_x x - D_x D_x x^2]_i \\ &= -2(1 + \frac{1}{2}t_n) \end{aligned}$$

Now,  $f_i^n = 2(1 + \frac{1}{2}t_n)c^2$  and we get

$$[D_t D_t u_e - c^2 D_x D_x u_e - f]_i^n = 0 - c^2(-1)2(1 + \frac{1}{2}t_n) + 2(1 + \frac{1}{2}t_n)c^2 = 0$$

Moreover,  $u_e(x_i, 0) = I(x_i)$ ,  $\partial u_e / \partial t = V(x_i)$  at  $t = 0$ , and  $u_e(x_0, t) = u_e(x_{N_x}, 0) = 0$ . Also the modified scheme for the first time step is fulfilled by  $u_e(x_i, t_n)$ .

### Testing with the exact discrete solution

- We have established that  $u_i^{n+1} = u_e(x_i, t_{n+1}) = x_i(L - x_i)(1 + t_{n+1}/2)$
- Run *one* simulation with one choice of  $c$ ,  $\Delta t$ , and  $\Delta x$
- Check that  $\max_i |u_i^{n+1} - u_e(x_i, t_{n+1})| < \epsilon$ ,  $\epsilon \sim 10^{-14}$  (machine precision + some round-off errors)
- This is the simplest and best verification test

Later we show that the exact solution of the discrete equations can be obtained by  $C = 1$  (!)

## Implementation

### The algorithm

1. Compute  $u_i^0 = I(x_i)$  for  $i = 0, \dots, N_x$
2. Compute  $u_i^1$  by (14) and set  $u_i^1 = 0$  for the boundary points  $i = 0$  and  $i = N_x$ , for  $n = 1, 2, \dots, N - 1$ ,
3. For each time level  $n = 1, 2, \dots, N_t - 1$ 
  - (a) apply (12) to find  $u_i^{n+1}$  for  $i = 1, \dots, N_x - 1$
  - (b) set  $u_i^{n+1} = 0$  for the boundary points  $i = 0, i = N_x$ .

## What do to with the solution?

- Different problem settings demand different actions with the computed  $u_i^{n+1}$  at each time step
- Solution: let the solver function make a callback to a user function where the user can do whatever is desired with the solution
- Advantage: solver just solves and user uses the solution

```
def user_action(u, x, t, n):  
    # u[i] at spatial mesh points x[i] at time t[n]  
    # plot u  
    # or store u
```

## Making a solver function (1)

We specify  $\Delta t$  and  $C$ , and let the solver function compute  $\Delta x = c\Delta t/C$ .

```
def solver(I, V, f, c, L, dt, C, T, user_action=None):  
    """Solve  $u_{tt}=c^2u_{xx} + f$  on  $(0,L) \times (0,T]$ ."""  
    Nt = int(round(T/dt))  
    t = linspace(0, Nt*dt, Nt+1) # Mesh points in time  
    dx = dt*c/float(C)  
    Nx = int(round(L/dx))  
    x = linspace(0, L, Nx+1)      # Mesh points in space  
    dx = x[1] - x[0]  
    C2 = C**2                      # Help variable in the scheme  
    if f is None or f == 0:  
        f = lambda x, t: 0  
    if V is None or V == 0:  
        V = lambda x: 0  
  
    u = zeros(Nx+1) # Solution array at new time level  
    u_1 = zeros(Nx+1) # Solution at 1 time level back  
    u_2 = zeros(Nx+1) # Solution at 2 time levels back  
  
    import time; t0 = time.clock() # for measuring CPU time  
  
    # Load initial condition into u_1  
    for i in range(0, Nx+1):  
        u_1[i] = I(x[i])  
  
    if user_action is not None:  
        user_action(u_1, x, t, 0)
```

## Making a solver function (2)

```
def solver(I, V, f, c, L, dt, C, T, user_action=None):  
    ...  
    # Special formula for first time step  
    n = 0  
    for i in range(1, Nx):
```

```

        u[i] = u_1[i] + dt*V(x[i]) + \
                0.5*C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
                0.5*dt**2*f(x[i], t[n])
    u[0] = 0; u[Nx] = 0

    if user_action is not None:
        user_action(u, x, t, 1)

    # Switch variables before next step
    u_2[:, u_1[:]] = u_1, u

```

## Making a solver function (3)

```

def solver(I, V, f, c, L, Nx, C, T, user_action=None):
    ...
    # Time loop

    for n in range(1, Nt):
        # Update all inner points at time t[n+1]
        for i in range(1, Nx):
            u[i] = - u_2[i] + 2*u_1[i] + \
                    C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
                    dt**2*f(x[i], t[n])

        # Insert boundary conditions
        u[0] = 0; u[Nx] = 0
        if user_action is not None:
            if user_action(u, x, t, n+1):
                break

        # Switch variables before next step
        u_2[:, u_1[:]] = u_1, u

    cpu_time = t0 - time.clock()
    return u, x, t, cpu_time

```

## Verification: exact quadratic solution

Exact solution of the PDE problem *and* the discrete equations:  $u_e(x, t) = x(L - x)(1 + \frac{1}{2}t)$

```

def test_quadratic():
    """Check that u(x,t)=x(L-x)(1+t/2) is exactly reproduced."""

    def u_exact(x, t):
        return x*(L-x)*(1 + 0.5*t)

    def I(x):
        return u_exact(x, 0)

    def V(x):
        return 0.5*u_exact(x, 0)

    def f(x, t):

```

```

        return 2*(1 + 0.5*t)*c**2

L = 2.5
c = 1.5
C = 0.75
Nx = 6 # Very coarse mesh for this exact test
dt = C*(L/Nx)/c
T = 18

def assert_no_error(u, x, t, n):
    u_e = u_exact(x, t[n])
    diff = np.abs(u - u_e).max()
    tol = 1E-13
    assert diff < tol

solver(I, V, f, c, L, dt, C, T,
       user_action=assert_no_error)

```

## Visualization: animating $u(x, t)$

Make a viz function for animating the curve, with plotting in a user\_action function plot\_u:

```

def viz(
    I, V, f, c, L, dt, C, T, # PDE paramteres
    umin, umax, # Interval for u in plots
    animate=True, # Simulation with animation?
    tool='matplotlib', # 'matplotlib' or 'scitools'
    solver_function=solver, # Function with numerical algorithm
):
    """Run solver and visualize u at each time level."""

    def plot_u_st(u, x, t, n):
        """user_action function for solver."""
        plt.plot(x, u, 'r-',
                 xlabel='x', ylabel='u',
                 axis=[0, L, umin, umax],
                 title='t=%f' % t[n], show=True)
        # Let the initial condition stay on the screen for 2
        # seconds, else insert a pause of 0.2 s between each plot
        time.sleep(2) if t[n] == 0 else time.sleep(0.2)
        plt.savefig('frame_%04d.png' % n) # for movie making

    class PlotMatplotlib:
        def __call__(self, u, x, t, n):
            """user_action function for solver."""
            if n == 0:
                plt.ion()
                self.lines = plt.plot(x, u, 'r-')
                plt.xlabel('x'); plt.ylabel('u')
                plt.axis([0, L, umin, umax])
                plt.legend(['t=%f' % t[n]], loc='lower left')
            else:
                self.lines[0].set_ydata(u)
                plt.legend(['t=%f' % t[n]], loc='lower left')
                plt.draw()

```

```

        time.sleep(2) if t[n] == 0 else time.sleep(0.2)
        plt.savefig('tmp_%04d.png' % n) # for movie making

    if tool == 'matplotlib':
        import matplotlib.pyplot as plt
        plot_u = PlotMatplotlib()
    elif tool == 'scitools':
        import scitools.std as plt # scitools.easyviz interface
        plot_u = plot_u_st
    import time, glob, os

    # Clean up old movie frames
    for filename in glob.glob('tmp_*.png'):
        os.remove(filename)

    # Call solver and do the simulation
    user_action = plot_u if animate else None
    u, x, t, cpu = solver_function(
        I, V, f, c, L, dt, C, T, user_action)

    # Make video files
    fps = 4 # frames per second
    codec2ext = dict(flv='flv', libx264='mp4', libvpx='webm',
                     libtheora='ogg') # video formats
    filespec = 'tmp_%04d.png'
    movie_program = 'ffmpeg' # or 'avconv'
    for codec in codec2ext:
        ext = codec2ext[codec]
        cmd = '%(movie_program)s -r %(fps)d -i %(filespec)s \' \
            -vcodec %(codec)s movie.%(ext)s' % vars()
        os.system(cmd)

    if tool == 'scitools':
        # Make an HTML play for showing the animation in a
        # browser
        plt.movie('tmp_*.png', encoder='html', fps=fps,
                  output_file='movie.html')

    return cpu

```

Note: `plot_u` is function inside function and remembers the local variables in `viz` (known as a closure).

## Making movie files

- Store spatial curve in a file, for each time level
- Name files like 'something\_%04d.png' % frame\_counter
- Combine files to a movie

```

Terminal> scitools movie encoder=html output_file=movie.html \
        fps=4 frame_*.png # web page with a player
Terminal> avconv -r 4 -i frame_%04d.png -c:v flv      movie.flv
Terminal> avconv -r 4 -i frame_%04d.png -c:v libtheora movie.ogg
Terminal> avconv -r 4 -i frame_%04d.png -c:v libx264  movie.mp4
Terminal> avconv -r 4 -i frame_%04d.png -c:v libvpx   movie.webm

```

### Important.

- Zero padding (%04d) is essential for correct sequence of frames in `something_*.png` (Unix alphanumeric sort)
- Remove old `frame_*.png` files before making a new movie

### Running a case

- Vibrations of a guitar string
- Triangular initial shape (at rest)

$$I(x) = \begin{cases} ax/x_0, & x < x_0 \\ a(L-x)/(L-x_0), & \text{otherwise} \end{cases} \quad (26)$$

Appropriate data:

- $L = 75$  cm,  $x_0 = 0.8L$ ,  $a = 5$  mm, time frequency  $\nu = 440$  Hz

### Implementation of the case

```
def guitar(C):
    """Triangular wave (pulled guitar string)."""
    L = 0.75
    x0 = 0.8*L
    a = 0.005
    freq = 440
    wavelength = 2*L
    c = freq*wavelength
    omega = 2*pi*freq
    num_periods = 1
    T = 2*pi/omega*num_periods
    # Choose dt the same as the stability limit for Nx=50
    dt = L/50./c

    def I(x):
        return a*x/x0 if x < x0 else a/(L-x0)*(L-x)

    umin = -1.2*a; umax = -umin
    cpu = viz(I, 0, 0, c, L, dt, C, T, umin, umax,
             animate=True, tool='scitools')
```

Program: `wave1D_u0.py`.

### Resulting movie for $C = 0.8$

Movie of the vibrating string

## The benefits of scaling

- It is difficult to figure out all the physical parameters of a case
- And it is not necessary because of a powerful: *scaling*

Introduce new  $x$ ,  $t$ , and  $u$  without dimension:

$$\bar{x} = \frac{x}{L}, \quad \bar{t} = \frac{c}{L}t, \quad \bar{u} = \frac{u}{a}$$

Insert this in the PDE (with  $f = 0$ ) and dropping bars

$$u_{tt} = u_{xx}$$

Initial condition: set  $a = 1$ ,  $L = 1$ , and  $x_0 \in [0, 1]$  in (26).

In the code: set `a=c=L=1`, `x0=0.8`, and there is no need to calculate with wavelengths and frequencies to estimate  $c$ !

Just one challenge: determine the period of the waves and an appropriate end time (see the text for details).

## Vectorization

- Problem: Python loops over long arrays are slow
- One remedy: use vectorized (`numpy`) code instead of explicit loops
- Other remedies: use Cython, port spatial loops to Fortran or C
- Speedup: 100-1000 (varies with  $N_x$ )

Next: vectorized loops

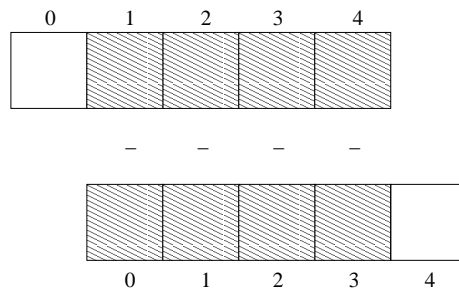
## Operations on slices of arrays

- Introductory example: compute  $d_i = u_{i+1} - u_i$

```
n = u.size
for i in range(0, n-1):
    d[i] = u[i+1] - u[i]
```

- Note: all the differences here are independent of each other.
- Therefore  $d = (u_1, u_2, \dots, u_n) - (u_0, u_1, \dots, u_{n-1})$
- In `numpy` code: `u[1:n] - u[0:n-1]` or just `u[1:] - u[:-1]`





## Test the understanding

Newcomers to vectorization are encouraged to choose a small array  $u$ , say with five elements, and simulate with pen and paper both the loop version and the vectorized version.

## Vectorization of finite difference schemes (1)

Finite difference schemes basically contains differences between array elements with shifted indices. Consider the updating formula

```
for i in range(1, n-1):
    u2[i] = u[i-1] - 2*u[i] + u[i+1]
```

The vectorization consists of replacing the loop by arithmetics on slices of arrays of length  $n-2$ :

```
u2 = u[:-2] - 2*u[1:-1] + u[2:]
u2 = u[0:n-2] - 2*u[1:n-1] + u[2:n] # alternative
```

Note:  $u2$  gets length  $n-2$ .

If  $u2$  is already an array of length  $n$ , do update on "inner" elements

```
u2[1:-1] = u[:-2] - 2*u[1:-1] + u[2:]
u2[1:n-1] = u[0:n-2] - 2*u[1:n-1] + u[2:n] # alternative
```

## Vectorization of finite difference schemes (2)

Include a function evaluation too:

```
def f(x):
    return x**2 + 1

# Scalar version
for i in range(1, n-1):
    u2[i] = u[i-1] - 2*u[i] + u[i+1] + f(x[i])

# Vectorized version
u2[1:-1] = u[:-2] - 2*u[1:-1] + u[2:] + f(x[1:-1])
```

## Vectorized implementation in the solver function

Scalar loop:

```
for i in range(1, Nx):
    u[i] = 2*u_1[i] - u_2[i] + \
          C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])
```

Vectorized loop:

```
u[1:-1] = - u_2[1:-1] + 2*u_1[1:-1] + \
          C2*(u_1[:-2] - 2*u_1[1:-1] + u_1[2:])
```

or

```
u[1:Nx] = 2*u_1[1:Nx] - u_2[1:Nx] + \
          C2*(u_1[0:Nx-1] - 2*u_1[1:Nx] + u_1[2:Nx+1])
```

Program: `wave1D_u0v.py`

## Verification of the vectorized version

```
def test_quadratic():
    """
    Check the scalar and vectorized versions work for
    a quadratic  $u(x,t)=x(L-x)(1+t/2)$  that is exactly reproduced.
    """
    # The following function must work for x as array or scalar
    u_exact = lambda x, t: x*(L - x)*(1 + 0.5*t)
    I = lambda x: u_exact(x, 0)
    V = lambda x: 0.5*u_exact(x, 0)
    # f is a scalar (zeros_like(x) works for scalar x too)
    f = lambda x, t: np.zeros_like(x) + 2*c**2*(1 + 0.5*t)

    L = 2.5
    c = 1.5
    C = 0.75
    Nx = 3 # Very coarse mesh for this exact test
    dt = C*(L/Nx)/c
    T = 18

    def assert_no_error(u, x, t, n):
        u_e = u_exact(x, t[n])
        tol = 1E-13
        diff = np.abs(u - u_e).max()
        assert diff < tol

    solver(I, V, f, c, L, dt, C, T,
           user_action=assert_no_error, version='scalar')
    solver(I, V, f, c, L, dt, C, T,
           user_action=assert_no_error, version='vectorized')
```

Note:

- Compact code with lambda functions
- The scalar  $f$  value needs careful coding: return constant array if vectorized code, else number

## Efficiency measurements

- Run `wave1D_u0v.py` for  $N_x$  as 50,100,200,400,800 and measuring the CPU time
- Observe substantial speed-up: vectorized version is about  $N_x/5$  times faster

Much bigger improvements for 2D and 3D codes!

## Generalization: reflecting boundaries

- Boundary condition  $u = 0$ :  $u$  changes sign
- Boundary condition  $u_x = 0$ : wave is perfectly reflected
- How can we implement  $u_x$ ? (more complicated than  $u = 0$ )

Demo of boundary conditions

## Neumann boundary condition

$$\frac{\partial u}{\partial n} \equiv \mathbf{n} \cdot \nabla u = 0 \quad (27)$$

For a 1D domain  $[0, L]$ :

$$\left. \frac{\partial}{\partial n} \right|_{x=L} = \frac{\partial}{\partial x}, \quad \left. \frac{\partial}{\partial n} \right|_{x=0} = -\frac{\partial}{\partial x}$$

Boundary condition terminology:

- $u_x$  specified: [Neumann](#) condition
- $u$  specified: [Dirichlet](#) condition

## Discretization of derivatives at the boundary (1)

- How can we incorporate the condition  $u_x = 0$  in the finite difference scheme?
- We used central differences for  $u_{tt}$  and  $u_{xx}$ :  $\mathcal{O}(\Delta t^2, \Delta x^2)$  accuracy
- Also for  $u_t(x, 0)$
- Should use central difference for  $u_x$  to preserve second order accuracy

$$\frac{u_{-1}^n - u_1^n}{2\Delta x} = 0 \quad (28)$$

## Discretization of derivatives at the boundary (2)

$$\frac{u_{-1}^n - u_1^n}{2\Delta x} = 0$$

- Problem:  $u_{-1}^n$  is outside the mesh (fictitious value)
- Remedy: use the stencil at the boundary to eliminate  $u_{-1}^n$ ; just replace  $u_{-1}^n$  by  $u_1^n$

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + 2C^2 (u_{i+1}^n - u_i^n), \quad i = 0 \quad (29)$$

## Visualization of modified boundary stencil

Discrete equation for computing  $u_0^3$  in terms of  $u_0^2$ ,  $u_0^1$ , and  $u_1^2$ :

Animation in a [web page](#) or a [movie file](#).

## Implementation of Neumann conditions

- Use the general stencil for interior points also on the boundary
- Replace  $u_{i-1}^n$  by  $u_{i+1}^n$  for  $i = 0$
- Replace  $u_{i+1}^n$  by  $u_{i-1}^n$  for  $i = N_x$

```
i = 0
ip1 = i+1
im1 = ip1 # i-1 -> i+1
u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])

i = Nx
im1 = i-1
ip1 = im1 # i+1 -> i-1
u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])

# Or just one loop over all points
for i in range(0, Nx+1):
    ip1 = i+1 if i < Nx else i-1
    im1 = i-1 if i > 0 else i+1
    u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])
```

Program [wave1D\\_dn0.py](#)

## Moving finite difference stencil

[web page](#) or a [movie file](#).

## Index set notation

- Tedious to write index sets like  $i = 0, \dots, N_x$  and  $n = 0, \dots, N_t$
- Notation not valid if  $i$  or  $n$  starts at 1 instead...
- Both in math and code it is advantageous to use *index sets*
- $i \in \mathcal{I}_x$  instead of  $i = 0, \dots, N_x$
- Definition:  $\mathcal{I}_x = \{0, \dots, N_x\}$
- The first index:  $i = \mathcal{I}_x^0$
- The last index:  $i = \mathcal{I}_x^{-1}$
- All interior points:  $i \in \mathcal{I}_x^i, \mathcal{I}_x^i = \{1, \dots, N_x - 1\}$
- $\mathcal{I}_x^-$  means  $\{0, \dots, N_x - 1\}$
- $\mathcal{I}_x^+$  means  $\{1, \dots, N_x\}$

## Index set notation in code

Notation	Python
$\mathcal{I}_x$	<code>Ix</code>
$\mathcal{I}_x^0$	<code>Ix[0]</code>
$\mathcal{I}_x^{-1}$	<code>Ix[-1]</code>
$\mathcal{I}_x^-$	<code>Ix[1:]</code>
$\mathcal{I}_x^+$	<code>Ix[:-1]</code>
$\mathcal{I}_x^i$	<code>Ix[1:-1]</code>

## Index sets in action (1)

Index sets for a problem in the  $x, t$  plane:

$$\mathcal{I}_x = \{0, \dots, N_x\}, \quad \mathcal{I}_t = \{0, \dots, N_t\}, \quad (30)$$

Implemented in Python as

```
Ix = range(0, Nx+1)
It = range(0, Nt+1)
```

## Index sets in action (2)

A finite difference scheme can with the index set notation be specified as

$$\begin{aligned} u_i^{n+1} &= -u_i^{n-1} + 2u_i^n + C^2 (u_{i+1}^n - 2u_i^n + u_{i-1}^n), \quad i \in \mathcal{I}_x^i, \quad n \in \mathcal{I}_t^i \\ u_i &= 0, \quad i = \mathcal{I}_x^0, \quad n \in \mathcal{I}_t^i \\ u_i &= 0, \quad i = \mathcal{I}_x^{-1}, \quad n \in \mathcal{I}_t^i \end{aligned}$$

Corresponding implementation:

```
for n in It[1:-1]:
    for i in Ix[1:-1]:
        u[i] = - u_2[i] + 2*u_1[i] + \
                C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])
    i = Ix[0]; u[i] = 0
    i = Ix[-1]; u[i] = 0
```

Program [wave1D\\_dn.py](#)

## Alternative implementation via ghost cells

- Instead of modifying the stencil at the boundary, we extend the mesh to cover  $u_{-1}^n$  and  $u_{N_x+1}^n$
- The extra left and right cell are called *ghost cells*
- The extra points are called *ghost points*
- The  $u_{-1}^n$  and  $u_{N_x+1}^n$  values are called *ghost values*
- Update ghost values as  $u_{i-1}^n = u_{i+1}^n$  for  $i = 0$  and  $i = N_x$
- Then the stencil becomes right at the boundary

## Implementation of ghost cells (1)

Add ghost points:

```
u = zeros(Nx+3)
u_1 = zeros(Nx+3)
u_2 = zeros(Nx+3)

x = linspace(0, L, Nx+1) # Mesh points without ghost points
```

- A major indexing problem arises with ghost cells since Python indices *must* start at 0.
- `u[-1]` will always mean the last element in `u`
- Math indexing:  $-1, 0, 1, 2, \dots, N_x + 1$

- Python indexing:  $0, \dots, Nx+2$
- Remedy: use index sets

## Implementation of ghost cells (2)

```

u = zeros(Nx+3)
Ix = range(1, u.shape[0]-1)

# Boundary values: u[Ix[0]], u[Ix[-1]]

# Set initial conditions
for i in Ix:
    u_1[i] = I(x[i-Ix[0]]) # Note i-Ix[0]

# Loop over all physical mesh points
for i in Ix:
    u[i] = - u_2[i] + 2*u_1[i] + \
           C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])

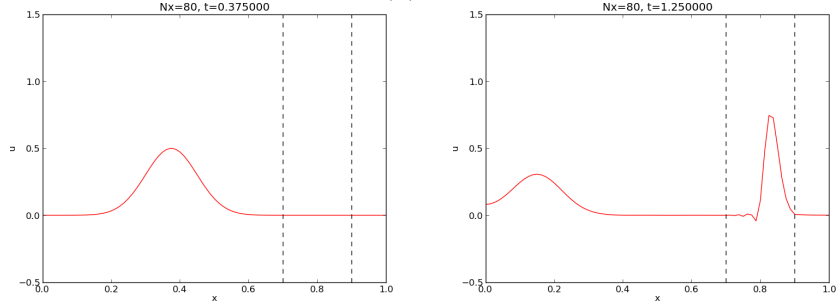
# Update ghost values
i = Ix[0] # x=0 boundary
u[i-1] = u[i+1]
i = Ix[-1] # x=L boundary
u[i-1] = u[i+1]

```

Program: [wave1D\\_dn0\\_ghost.py](#).

## Generalization: variable wave velocity

Heterogeneous media: varying  $c = c(x)$



## The model PDE with a variable coefficient

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x} \left( q(x) \frac{\partial u}{\partial x} \right) + f(x, t) \quad (31)$$

This equation sampled at a mesh point  $(x_i, t_n)$ :

$$\frac{\partial^2}{\partial t^2} u(x_i, t_n) = \frac{\partial}{\partial x} \left( q(x_i) \frac{\partial}{\partial x} u(x_i, t_n) \right) + f(x_i, t_n),$$

### Discretizing the variable coefficient (1)

The principal idea is to *first discretize the outer derivative*.

Define

$$\phi = q(x) \frac{\partial u}{\partial x}$$

Then use a centered derivative around  $x = x_i$  for the derivative of  $\phi$ :

$$\left[ \frac{\partial \phi}{\partial x} \right]_i^n \approx \frac{\phi_{i+\frac{1}{2}} - \phi_{i-\frac{1}{2}}}{\Delta x} = [D_x \phi]_i^n$$

### Discretizing the variable coefficient (2)

Then discretize the inner operators:

$$\phi_{i+\frac{1}{2}} = q_{i+\frac{1}{2}} \left[ \frac{\partial u}{\partial x} \right]_{i+\frac{1}{2}}^n \approx q_{i+\frac{1}{2}} \frac{u_{i+1}^n - u_i^n}{\Delta x} = [q D_x u]_{i+\frac{1}{2}}^n$$

Similarly,

$$\phi_{i-\frac{1}{2}} = q_{i-\frac{1}{2}} \left[ \frac{\partial u}{\partial x} \right]_{i-\frac{1}{2}}^n \approx q_{i-\frac{1}{2}} \frac{u_i^n - u_{i-1}^n}{\Delta x} = [q D_x u]_{i-\frac{1}{2}}^n$$

### Discretizing the variable coefficient (3)

These intermediate results are now combined to

$$\left[ \frac{\partial}{\partial x} \left( q(x) \frac{\partial u}{\partial x} \right) \right]_i^n \approx \frac{1}{\Delta x^2} \left( q_{i+\frac{1}{2}} (u_{i+1}^n - u_i^n) - q_{i-\frac{1}{2}} (u_i^n - u_{i-1}^n) \right) \quad (32)$$

In operator notation:

$$\left[ \frac{\partial}{\partial x} \left( q(x) \frac{\partial u}{\partial x} \right) \right]_i^n \approx [D_x q D_x u]_i^n \quad (33)$$

**Remark.** Many are tempted to use the chain rule on the term  $\frac{\partial}{\partial x} \left( q(x) \frac{\partial u}{\partial x} \right)$ , but this is not a good idea!

### Computing the coefficient between mesh points

- Given  $q(x)$ : compute  $q_{i+\frac{1}{2}}$  as  $q(x_{i+\frac{1}{2}})$
- Given  $q$  at the mesh points:  $q_i$ , use an average



$$q_{i+\frac{1}{2}} \approx \frac{1}{2} (q_i + q_{i+1}) = [\bar{q}^x]_i \quad (\text{arithmetic mean}) \quad (34)$$

$$q_{i+\frac{1}{2}} \approx 2 \left( \frac{1}{q_i} + \frac{1}{q_{i+1}} \right)^{-1} \quad (\text{harmonic mean}) \quad (35)$$

$$q_{i+\frac{1}{2}} \approx (q_i q_{i+1})^{1/2} \quad (\text{geometric mean}) \quad (36)$$

The arithmetic mean in (34) is by far the most used averaging technique.

## Discretization of variable-coefficient wave equation in operator notation

$$[D_t D_t u = D_x \bar{q}^x D_x u + f]_i^n \quad (37)$$

We clearly see the type of finite differences and averaging!

Write out and solve wrt  $u_i^{n+1}$ :

$$\begin{aligned} u_i^{n+1} = & -u_i^{n-1} + 2u_i^n + \left( \frac{\Delta x}{\Delta t} \right)^2 \times \\ & \left( \frac{1}{2} (q_i + q_{i+1}) (u_{i+1}^n - u_i^n) - \frac{1}{2} (q_i + q_{i-1}) (u_i^n - u_{i-1}^n) \right) + \\ & \Delta t^2 f_i^n \end{aligned} \quad (38)$$

## Neumann condition and a variable coefficient

Consider  $\partial u / \partial x = 0$  at  $x = L = N_x \Delta x$ :

$$\frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} = 0 \quad u_{i+1}^n = u_{i-1}^n, \quad i = N_x$$

Insert  $u_{i+1}^n = u_{i-1}^n$  in the stencil (38) for  $i = N_x$  and obtain

$$u_i^{n+1} \approx -u_i^{n-1} + 2u_i^n + \left( \frac{\Delta x}{\Delta t} \right)^2 2q_i (u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n$$

(We have used  $q_{i+\frac{1}{2}} + q_{i-\frac{1}{2}} \approx 2q_i$ .)

Alternative: assume  $dq/dx = 0$  (simpler).

## Implementation of variable coefficients

Assume  $c[i]$  holds  $c_i$  the spatial mesh points

```
for i in range(1, Nx):
    u[i] = - u_2[i] + 2*u_1[i] + \
        c2*(0.5*(q[i] + q[i+1])*(u_1[i+1] - u_1[i]) - \
            0.5*(q[i] + q[i-1])*(u_1[i] - u_1[i-1])) + \
        dt2*f(x[i], t[n])
```

Here:  $C2=(dt/dx)**2$

Vectorized version:

```
u[1:-1] = - u_2[1:-1] + 2*u_1[1:-1] + \
           C2*(0.5*(q[1:-1] + q[2:])*(u_1[2:] - u_1[1:-1]) -
              0.5*(q[1:-1] + q[:-2])*(u_1[1:-1] - u_1[:-2])) + \
           dt2*f(x[1:-1], t[n])
```

Neumann condition  $u_x = 0$ : same ideas as in 1D (modified stencil or ghost cells).

### A more general model PDE with variable coefficients

$$\varrho(x) \frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x} \left( q(x) \frac{\partial u}{\partial x} \right) + f(x, t) \quad (39)$$

A natural scheme is

$$[\varrho D_t D_t u = D_x \bar{q}^x D_x u + f]_i^n \quad (40)$$

Or

$$[D_t D_t u = \varrho^{-1} D_x \bar{q}^x D_x u + f]_i^n \quad (41)$$

No need to average  $\varrho$ , just sample at  $i$

### Generalization: damping

Why do waves die out?

- Damping (non-elastic effects, air resistance)
- 2D/3D: conservation of energy makes an amplitude reduction by  $1/\sqrt{r}$  (2D) or  $1/r$  (3D)

Simplest damping model (for physical behavior, see [demo](#)):

$$\frac{\partial^2 u}{\partial t^2} + b \frac{\partial u}{\partial t} = c^2 \frac{\partial^2 u}{\partial x^2} + f(x, t), \quad (42)$$

$b \geq 0$ : prescribed damping coefficient.

Discretization via centered differences to ensure  $\mathcal{O}(\Delta t^2)$  error:

$$[D_t D_t u + b D_{2t} u = c^2 D_x D_x u + f]_i^n \quad (43)$$

Need special formula for  $u_i^1$  + special stencil (or ghost cells) for Neumann conditions.

## Building a general 1D wave equation solver

The program `wave1D_dn_vc.py` solves a fairly general 1D wave equation:

$$u_t = (c^2(x)u_x)_x + f(x, t), \quad x \in (0, L), \quad t \in (0, T] \quad (44)$$

$$u(x, 0) = I(x), \quad x \in [0, L] \quad (45)$$

$$u_t(x, 0) = V(t), \quad x \in [0, L] \quad (46)$$

$$u(0, t) = U_0(t) \text{ or } u_x(0, t) = 0, \quad t \in (0, T] \quad (47)$$

$$u(L, t) = U_L(t) \text{ or } u_x(L, t) = 0, \quad t \in (0, T] \quad (48)$$

Can be adapted to many needs.

### Collection of initial conditions

The function `pulse` in `wave1D_dn_vc.py` offers four initial conditions:

1. a rectangular pulse ("plug")
2. a Gaussian function (`gaussian`)
3. a "cosine hat": one period of  $1 + \cos(\pi x)$ ,  $x \in [-1, 1]$
4. half a "cosine hat": half a period of  $\cos \pi x$ ,  $x \in [-\frac{1}{2}, \frac{1}{2}]$

Can locate the initial pulse at  $x = 0$  or in the middle

```
>> import wave1D_dn_vc as w
>> w.pulse(loc='left', pulse_tp='cosinehat', Nx=50,
          every_frame=10)
```

## Finite difference methods for 2D and 3D wave equations

Constant wave velocity  $c$ :

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \text{ for } \mathbf{x} \in \Omega \subset \mathbb{R}^d, \quad t \in (0, T] \quad (49)$$

Variable wave velocity:

$$\varrho \frac{\partial^2 u}{\partial t^2} = \nabla \cdot (q \nabla u) + f \text{ for } \mathbf{x} \in \Omega \subset \mathbb{R}^d, \quad t \in (0, T] \quad (50)$$

## Examples on wave equations written out in 2D/3D

3D, constant  $c$ :

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$$

2D, variable  $c$ :

$$\varrho(x, y) \frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x} \left( q(x, y) \frac{\partial u}{\partial x} \right) + \frac{\partial}{\partial y} \left( q(x, y) \frac{\partial u}{\partial y} \right) + f(x, y, t) \quad (51)$$

Compact notation:

$$u_{tt} = c^2(u_{xx} + u_{yy} + u_{zz}) + f, \quad (52)$$

$$\varrho u_{tt} = (qu_x)_x + (qu_z)_z + f \quad (53)$$

## Boundary and initial conditions

We need *one* boundary condition at *each point* on  $\partial\Omega$ :

1.  $u$  is prescribed ( $u = 0$  or known incoming wave)
2.  $\partial u / \partial n = \mathbf{n} \cdot \nabla u$  prescribed ( $= 0$ : reflecting boundary)
3. open boundary (radiation) condition:  $u_t + \mathbf{c} \cdot \nabla u = 0$  (let waves travel undisturbed out of the domain)

PDEs with *second-order* time derivative need *two* initial conditions:

1.  $u = I$ ,
2.  $u_t = V$ .

## Example: 2D propagation of Gaussian function

### Mesh

- Mesh point:  $(x_i, y_j, z_k, t_n)$
- $x$  direction:  $x_0 < x_1 < \dots < x_{N_x}$
- $y$  direction:  $y_0 < y_1 < \dots < y_{N_y}$
- $z$  direction:  $z_0 < z_1 < \dots < z_{N_z}$
- $u_{i,j,k}^n \approx u_e(x_i, y_j, z_k, t_n)$

## Discretization

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u) + f]_{i,j,k}^n,$$

Written out in detail:

$$\frac{u_{i,j}^{n+1} - 2u_{i,j}^n + u_{i,j}^{n-1}}{\Delta t^2} = c^2 \frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{\Delta x^2} + c^2 \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{\Delta y^2} + f_{i,j}^n,$$

$u_{i,j}^{n-1}$  and  $u_{i,j}^n$  are known, solve for  $u_{i,j}^{n+1}$ :

$$u_{i,j}^{n+1} = 2u_{i,j}^n + u_{i,j}^{n-1} + c^2 \Delta t^2 [D_x D_x u + D_y D_y u]_{i,j}^n$$

## Special stencil for the first time step

- The stencil for  $u_{i,j}^1$  ( $n = 0$ ) involves  $u_{i,j}^{-1}$  which is outside the time mesh
- Remedy: use discretized  $u_t(x, 0) = V$  and the stencil for  $n = 0$  to develop a special stencil (as in the 1D case)

$$[D_{2t} u = V]_{i,j}^0 \Rightarrow u_{i,j}^{-1} = u_{i,j}^1 - 2\Delta t V_{i,j}$$

$$u_{i,j}^{n+1} = u_{i,j}^n - 2\Delta V_{i,j} + \frac{1}{2} c^2 \Delta t^2 [D_x D_x u + D_y D_y u]_{i,j}^n$$

## Variable coefficients (1)

3D wave equation:

$$\varrho u_{tt} = (\varrho u_x)_x + (\varrho u_y)_y + (\varrho u_z)_z + f(x, y, z, t)$$

Just apply the 1D discretization for each term:

$$[\varrho D_t D_t u = (D_x \bar{q}^x D_x u + D_y \bar{q}^y D_y u + D_z \bar{q}^z D_z u) + f]_{i,j,k}^n \quad (54)$$

Need special formula for  $u_{i,j,k}^1$  (use  $[D_{2t} u = V]^0$  and stencil for  $n = 0$ ).

## Variable coefficients (2)

Written out:

$$\begin{aligned}
u_{i,j,k}^{n+1} &= -u_{i,j,k}^{n-1} + 2u_{i,j,k}^n + \\
&= \frac{1}{\varrho_{i,j,k}} \frac{1}{\Delta x^2} \left( \frac{1}{2} (q_{i,j,k} + q_{i+1,j,k}) (u_{i+1,j,k}^n - u_{i,j,k}^n) - \right. \\
&\quad \left. \frac{1}{2} (q_{i-1,j,k} + q_{i,j,k}) (u_{i,j,k}^n - u_{i-1,j,k}^n) \right) + \\
&= \frac{1}{\varrho_{i,j,k}} \frac{1}{\Delta x^2} \left( \frac{1}{2} (q_{i,j,k} + q_{i,j+1,k}) (u_{i,j+1,k}^n - u_{i,j,k}^n) - \right. \\
&\quad \left. \frac{1}{2} (q_{i,j-1,k} + q_{i,j,k}) (u_{i,j,k}^n - u_{i,j-1,k}^n) \right) + \\
&= \frac{1}{\varrho_{i,j,k}} \frac{1}{\Delta x^2} \left( \frac{1}{2} (q_{i,j,k} + q_{i,j,k+1}) (u_{i,j,k+1}^n - u_{i,j,k}^n) - \right. \\
&\quad \left. \frac{1}{2} (q_{i,j,k-1} + q_{i,j,k}) (u_{i,j,k}^n - u_{i,j,k-1}^n) \right) + \\
&+ \Delta t^2 f_{i,j,k}^n
\end{aligned}$$

## Neumann boundary condition in 2D

Use ideas from 1D! Example:  $\frac{\partial u}{\partial n}$  at  $y = 0$ ,  $\frac{\partial u}{\partial n} = -\frac{\partial u}{\partial y}$

Boundary condition discretization:

$$[-D_{2y}u = 0]_{i,0}^n \Rightarrow \frac{u_{i,1}^n - u_{i,-1}^n}{2\Delta y} = 0, \quad i \in \mathcal{I}_x$$

Insert  $u_{i,-1}^n = u_{i,1}^n$  in the stencil for  $u_{i,j=0}^{n+1}$  to obtain a modified stencil on the boundary.

Pattern: use interior stencil also on the boundary, but replace  $j - 1$  by  $j + 1$

Alternative: use ghost cells and ghost values

## Implementation of 2D/3D problems

$$u_t = c^2(u_{xx} + u_{yy}) + f(x, y, t), \quad (x, y) \in \Omega, \quad t \in (0, T] \quad (55)$$

$$u(x, y, 0) = I(x, y), \quad (x, y) \in \Omega \quad (56)$$

$$u_t(x, y, 0) = V(x, y), \quad (x, y) \in \Omega \quad (57)$$

$$u = 0, \quad (x, y) \in \partial\Omega, \quad t \in (0, T] \quad (58)$$

$$\Omega = [0, L_x] \times [0, L_y]$$

Discretization:

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u) + f]_{i,j}^n,$$

## Algorithm

1. Set initial condition  $u_{i,j}^0 = I(x_i, y_j)$
2. Compute  $u_{i,j}^1 = \dots$  for  $i \in \mathcal{I}_x^i$  and  $j \in \mathcal{I}_y^i$
3. Set  $u_{i,j}^1 = 0$  for the boundaries  $i = 0, N_x, j = 0, N_y$
4. For  $n = 1, 2, \dots, N_t$ :
  - (a) Find  $u_{i,j}^{n+1} = \dots$  for  $i \in \mathcal{I}_x^i$  and  $j \in \mathcal{I}_y^i$
  - (b) Set  $u_{i,j}^{n+1} = 0$  for the boundaries  $i = 0, N_x, j = 0, N_y$

## Scalar computations: mesh

Program: `wave2D_u0.py`

```
def solver(I, V, f, c, Lx, Ly, Nx, Ny, dt, T,
           user_action=None, version='scalar'):
```

Mesh:

```
x = linspace(0, Lx, Nx+1)           # mesh points in x dir
y = linspace(0, Ly, Ny+1)           # mesh points in y dir
dx = x[1] - x[0]
dy = y[1] - y[0]
Nt = int(round(T/float(dt)))
t = linspace(0, N*dt, N+1)          # mesh points in time
Cx2 = (c*dt/dx)**2; Cy2 = (c*dt/dy)**2 # help variables
dt2 = dt**2
```

## Scalar computations: arrays

Store  $u_{i,j}^{n+1}$ ,  $u_{i,j}^n$ , and  $u_{i,j}^{n-1}$  in three two-dimensional arrays:

```
u = zeros((Nx+1, Ny+1))             # solution array
u_1 = zeros((Nx+1, Ny+1))           # solution at t-dt
u_2 = zeros((Nx+1, Ny+1))           # solution at t-2*dt
```

$u_{i,j}^{n+1}$  corresponds to `u[i,j]`, etc.

## Scalar computations: initial condition

```
Ix = range(0, u.shape[0])
Iy = range(0, u.shape[1])
It = range(0, t.shape[0])

for i in Ix:
    for j in Iy:
        u_1[i,j] = I(x[i], y[j])

if user_action is not None:
    user_action(u_1, x, xv, y, yv, t, 0)
```

Arguments `xv` and `yv`: for vectorized computations

## Scalar computations: primary stencil

```
def advance_scalar(u, u_1, u_2, f, x, y, t, n, Cx2, Cy2, dt2,
                  V=None, step1=False):
    Ix = range(0, u.shape[0]); Iy = range(0, u.shape[1])
    if step1:
        dt = sqrt(dt2) # save
        Cx2 = 0.5*Cx2; Cy2 = 0.5*Cy2; dt2 = 0.5*dt2 # redefine
        D1 = 1; D2 = 0
    else:
        D1 = 2; D2 = 1
    for i in Ix[1:-1]:
        for j in Iy[1:-1]:
            u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
            u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
            u[i,j] = D1*u_1[i,j] - D2*u_2[i,j] + \
                    Cx2*u_xx + Cy2*u_yy + dt2*f(x[i], y[j],
                    t[n])
            if step1:
                u[i,j] += dt*V(x[i], y[j])
    # Boundary condition u=0
    j = Iy[0]
    for i in Ix: u[i,j] = 0
    j = Iy[-1]
    for i in Ix: u[i,j] = 0
    i = Ix[0]
    for j in Iy: u[i,j] = 0
    i = Ix[-1]
    for j in Iy: u[i,j] = 0
    return u
```

D1 and D2: allow advance\_scalar to be used also for  $u_{i,j}^1$ :

```
u = advance_scalar(u, u_1, u_2, f, x, y, t,
                  n, 0.5*Cx2, 0.5*Cy2, 0.5*dt2, D1=1, D2=0)
```

## Vectorized computations: mesh coordinates

Mesh with  $30 \times 30$  cells: vectorization reduces the CPU time by a factor of 70 (!).

Need special coordinate arrays xv and yv such that  $I(x, y)$  and  $f(x, y, t)$  can be vectorized:

```
from numpy import newaxis
xv = x[:,newaxis]
yv = y[newaxis,:]

u_1[:, :] = I(xv, yv)
f_a[:, :] = f(xv, yv, t)
```

## Vectorized computations: stencil

```
def advance_vectorized(u, u_1, u_2, f_a, Cx2, Cy2, dt2,
                      V=None, step1=False):
```



```

if step1:
    dt = sqrt(dt2) # save
    Cx2 = 0.5*Cx2; Cy2 = 0.5*Cy2; dt2 = 0.5*dt2 # redefine
    D1 = 1; D2 = 0
else:
    D1 = 2; D2 = 1
u_xx = u_1[:-2,1:-1] - 2*u_1[1:-1,1:-1] + u_1[2:,1:-1]
u_yy = u_1[1:-1,:-2] - 2*u_1[1:-1,1:-1] + u_1[1:-1,2:]
u[1:-1,1:-1] = D1*u_1[1:-1,1:-1] - D2*u_2[1:-1,1:-1] + \
    Cx2*u_xx + Cy2*u_yy + dt2*f_a[1:-1,1:-1]

if step1:
    u[1:-1,1:-1] += dt*V[1:-1, 1:-1]
# Boundary condition u=0
j = 0
u[:,j] = 0
j = u.shape[1]-1
u[:,j] = 0
i = 0
u[i,:] = 0
i = u.shape[0]-1
u[i,:] = 0
return u

def quadratic(Nx, Ny, version):
    """Exact discrete solution of the scheme."""

    def exact_solution(x, y, t):
        return x*(Lx - x)*y*(Ly - y)*(1 + 0.5*t)

    def I(x, y):
        return exact_solution(x, y, 0)

    def V(x, y):
        return 0.5*exact_solution(x, y, 0)

    def f(x, y, t):
        return 2*c**2*(1 + 0.5*t)*(y*(Ly - y) + x*(Lx - x))

    Lx = 5; Ly = 2
    c = 1.5
    dt = -1 # use longest possible steps
    T = 18

    def assert_no_error(u, x, xv, y, yv, t, n):
        u_e = exact_solution(xv, yv, t[n])
        diff = abs(u - u_e).max()
        tol = 1E-12
        msg = 'diff=%g, step %d, time=%g' % (diff, n, t[n])
        assert diff < tol, msg

    new_dt, cpu = solver(
        I, V, f, c, Lx, Ly, Nx, Ny, dt, T,
        user_action=assert_no_error, version=version)
    return new_dt, cpu

def test_quadratic():

```

```

# Test a series of meshes where Nx > Ny and Nx < Ny
versions = 'scalar', 'vectorized', 'cython', 'f77', 'c_cy',
           'c_f2py'
for Nx in range(2, 6, 2):
    for Ny in range(2, 6, 2):
        for version in versions:
            print 'testing', version, 'for %dx%d mesh' %
                (Nx, Ny)
            quadratic(Nx, Ny, version)

def run_efficiency(nrefinements=4):
    def I(x, y):
        return sin(pi*x/Lx)*sin(pi*y/Ly)

    Lx = 10; Ly = 10
    c = 1.5
    T = 100
    versions = ['scalar', 'vectorized', 'cython', 'f77',
                'c_f2py', 'c_cy']
    print ' '*15, ''.join(['%-13s' % v for v in versions])
    for Nx in 15, 30, 60, 120:
        cpu = {}
        for version in versions:
            dt, cpu_ = solver(I, None, None, c, Lx, Ly, Nx, Nx,
                              -1, T, user_action=None,
                              version=version)

            cpu[version] = cpu_
        cpu_min = min(list(cpu.values()))
        if cpu_min < 1E-6:
            print 'Ignored %dx%d grid (too small execution
                  time)' \
                  % (Nx, Nx)
        else:
            cpu = {version: cpu[version]/cpu_min for version in
                    cpu}
            print '%-15s' % '%dx%d' % (Nx, Nx),
            print ''.join(['%13.1f' % cpu[version] for version
                            in versions])

def gaussian(plot_method=2, version='vectorized',
             save_plot=True):
    """
    Initial Gaussian bell in the middle of the domain.
    plot_method=1 applies mesh function, =2 means surf, =0 means
    no plot.
    """
    # Clean up plot files
    for name in glob('tmp_*.png'):
        os.remove(name)

    Lx = 10
    Ly = 10
    c = 1.0

    def I(x, y):
        """Gaussian peak at (Lx/2, Ly/2)."""
        return exp(-0.5*(x-Lx/2.0)**2 - 0.5*(y-Ly/2.0)**2)

```

```

if plot_method == 3:
    from mpl_toolkits.mplot3d import axes3d
    import matplotlib.pyplot as plt
    from matplotlib import cm
    plt.ion()
    fig = plt.figure()
    u_surf = None

def plot_u(u, x, xv, y, yv, t, n):
    if t[n] == 0:
        time.sleep(2)
    if plot_method == 1:
        mesh(x, y, u, title='t=%g' % t[n], zlim=[-1,1],
             caxis=[-1,1])
    elif plot_method == 2:
        surfc(xv, yv, u, title='t=%g' % t[n], zlim=[-1, 1],
             colorbar=True, colormap=hot(), caxis=[-1,1],
             shading='flat')
    elif plot_method == 3:
        print 'Experimental 3D matplotlib...under
              development...'
        #plt.clf()
        ax = fig.add_subplot(111, projection='3d')
        u_surf = ax.plot_surface(xv, yv, u, alpha=0.3)
        #ax.contourf(xv, yv, u, zdir='z', offset=-100,
        #            cmap=cm.coolwarm)
        #ax.set_zlim(-1, 1)
        # Remove old surface before drawing
        if u_surf is not None:
            ax.collections.remove(u_surf)
        plt.draw()
        time.sleep(1)
    if plot_method > 0:
        time.sleep(0) # pause between frames
        if save_plot:
            filename = 'tmp_%04d.png' % n
            savefig(filename) # time consuming!

Nx = 40; Ny = 40; T = 20
dt, cpu = solver(I, None, None, c, Lx, Ly, Nx, Ny, -1, T,
                 user_action=plot_u, version=version)

if __name__ == '__main__':
    test_quadratic()

```

## Verification: quadratic solution (1)

Manufactured solution:

$$u_e(x, y, t) = x(L_x - x)y(L_y - y)(1 + \frac{1}{2}t) \quad (59)$$

Requires  $f = 2c^2(1 + \frac{1}{2}t)(y(L_y - y) + x(L_x - x))$ .

This  $u_e$  is ideal because it also solves the discrete equations!

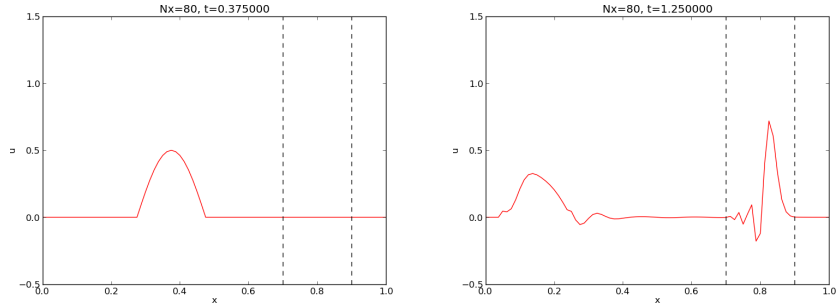
### Verification: quadratic solution (2)

- $[D_t D_t 1]^n = 0$
- $[D_t D_t t]^n = 0$
- $[D_t D_t t^2] = 2$
- $D_t D_t$  is a linear operator:  $[D_t D_t (au + bv)]^n = a[D_t D_t u]^n + b[D_t D_t v]^n$

$$\begin{aligned} [D_x D_x u_e]_{i,j}^n &= [y(L_y - y)(1 + \frac{1}{2}t)D_x D_x x(L_x - x)]_{i,j}^n \\ &= y_j(L_y - y_j)(1 + \frac{1}{2}t_n)2 \end{aligned}$$

- Similar calculations for  $[D_y D_y u_e]_{i,j}^n$  and  $[D_t D_t u_e]_{i,j}^n$  terms
- Must also check the equation for  $u_{i,j}^1$

### Analysis of the difference equations



### Properties of the solution of the wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

Solutions:

$$u(x, t) = g_R(x - ct) + g_L(x + ct)$$

If  $u(x, 0) = I(x)$  and  $u_t(x, 0) = 0$ :

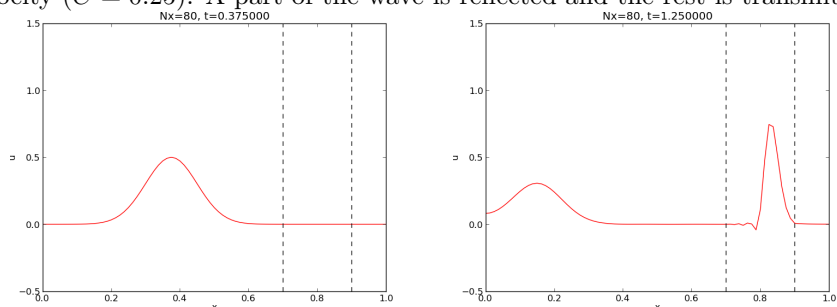
$$u(x, t) = \frac{1}{2}I(x - ct) + \frac{1}{2}I(x + ct)$$

Two waves: one traveling to the right and one to the left

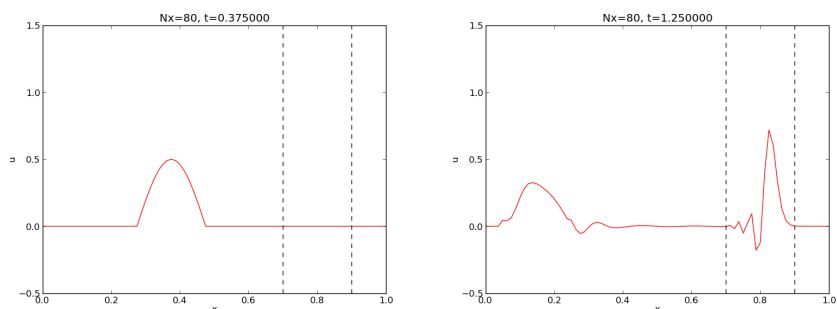
## Demo of the splitting of $I(x)$ into two waves

### Simulation of a case with variable wave velocity

A wave propagates perfectly ( $C = 1$ ) and hits a medium with  $1/4$  of the wave velocity ( $C = 0.25$ ). A part of the wave is reflected and the rest is transmitted.



Let us change the shape of the initial condition slightly and see what happens



### Representation of waves as sum of sine/cosine waves

Build  $I(x)$  of wave components  $e^{ikx} = \cos kx + i \sin kx$ :

$$I(x) \approx \sum_{k \in K} b_k e^{ikx}$$

- Fit  $b_k$  by a least squares or projection method
- $k$  is the frequency of a component ( $\lambda = 2\pi/k$  is the wave length in space)
- $K$  is some set of all  $k$  needed to approximate  $I(x)$  well
- $b_k$  must be computed (Fourier coefficients)

Since  $u(x, t) = \frac{1}{2}I(x - ct) + \frac{1}{2}I(x + ct)$ , the exact solution is

$$u(x, t) = \frac{1}{2} \sum_{k \in K} b_k e^{ik(x-ct)} + \frac{1}{2} \sum_{k \in K} b_k e^{ik(x+ct)}$$

Our interest: one component  $e^{i(kx-\omega t)}$ ,  $\omega = kc$

## A similar wave component is also a solution of the finite difference scheme (!)

Idea: a similar discrete  $u_q^n = e^{i(kx_q - \tilde{\omega}t_n)}$  solution (corresponding to the exact  $e^{i(kx - \omega t)}$ ) solves

$$[D_t D_t u = c^2 D_x D_x u]_q^n$$

Note: we expect numerical frequency  $\tilde{\omega} \neq \omega$

- How accurate is  $\tilde{\omega}$  compared to  $\omega$ ?
- What about the wave amplitude (can  $\tilde{\omega}$  become complex)?

## Preliminary results

$$[D_t D_t e^{i\omega t}]^n = -\frac{4}{\Delta t^2} \sin^2\left(\frac{\omega \Delta t}{2}\right) e^{i\omega n \Delta t}$$

By  $\omega \rightarrow k$ ,  $t \rightarrow x$ ,  $n \rightarrow q$ ) it follows that

$$[D_x D_x e^{ikx}]_q = -\frac{4}{\Delta x^2} \sin^2\left(\frac{k \Delta x}{2}\right) e^{ikq \Delta x}$$

## Insertion of the numerical wave component

Inserting a basic wave component  $u = e^{i(kx_q - \tilde{\omega}t_n)}$  in the scheme requires computation of

$$\begin{aligned} [D_t D_t e^{ikx} e^{-i\tilde{\omega}t}]_q^n &= [D_t D_t e^{-i\tilde{\omega}t}]^n e^{ikq \Delta x} \\ &= -\frac{4}{\Delta t^2} \sin^2\left(\frac{\tilde{\omega} \Delta t}{2}\right) e^{-i\tilde{\omega} n \Delta t} e^{ikq \Delta x} \\ [D_x D_x e^{ikx} e^{-i\tilde{\omega}t}]_q^n &= [D_x D_x e^{ikx}]_q e^{-i\tilde{\omega} n \Delta t} \\ &= -\frac{4}{\Delta x^2} \sin^2\left(\frac{k \Delta x}{2}\right) e^{ikq \Delta x} e^{-i\tilde{\omega} n \Delta t} \end{aligned}$$

## The equation for $\tilde{\omega}$

The complete scheme,

$$[D_t D_t e^{ikx} e^{-i\tilde{\omega}t} = c^2 D_x D_x e^{ikx} e^{-i\tilde{\omega}t}]_q^n$$

leads to an equation for  $\tilde{\omega}$  (which can readily be solved):

$$\sin^2\left(\frac{\tilde{\omega} \Delta t}{2}\right) = C^2 \sin^2\left(\frac{k \Delta x}{2}\right), \quad C = \frac{c \Delta t}{\Delta x} \text{ (Courant number)}$$

Taking the square root:

$$\sin\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C \sin\left(\frac{k\Delta x}{2}\right)$$

## The numerical dispersion relation

Can easily solve for an explicit formula for  $\tilde{\omega}$ :

$$\tilde{\omega} = \frac{2}{\Delta t} \sin^{-1}\left(C \sin\left(\frac{k\Delta x}{2}\right)\right)$$

Note:

- This  $\tilde{\omega} = \tilde{\omega}(k, c, \Delta x, \Delta t)$  is the *numerical dispersion relation*
- Inserting  $e^{kx-\omega t}$  in the PDE leads to  $\omega = kc$ , which is the *analytical/exact dispersion relation*
- Speed of waves might be easier to imagine:
  - Exact speed:  $c = \omega/k$ ,
  - Numerical speed:  $\tilde{c} = \tilde{\omega}/k$
- We shall investigate  $\tilde{c}/c$  to see how wrong the speed of a numerical wave component is

## The special case $C = 1$ gives the exact solution

- For  $C = 1$ ,  $\tilde{\omega} = \omega$
- The numerical solution is exact (at the mesh points), regardless of  $\Delta x$  and  $\Delta t = c^{-1}\Delta x$ !
- The only requirement is constant  $c$
- The numerical scheme is then a simple-to-use analytical solution method for the wave equation

## Computing the error in wave velocity

- Introduce  $p = k\Delta x/2$   
(the important *dimensionless* spatial discretization parameter)
- $p$  measures no of mesh points in space per wave length in space
- Shortest possible wave length in mesh:  $\lambda = 2\Delta x$ ,  $k = 2\pi/\lambda = \pi/\Delta x$ , and  $p = k\Delta x/2 = \pi/2 \Rightarrow p \in (0, \pi/2]$

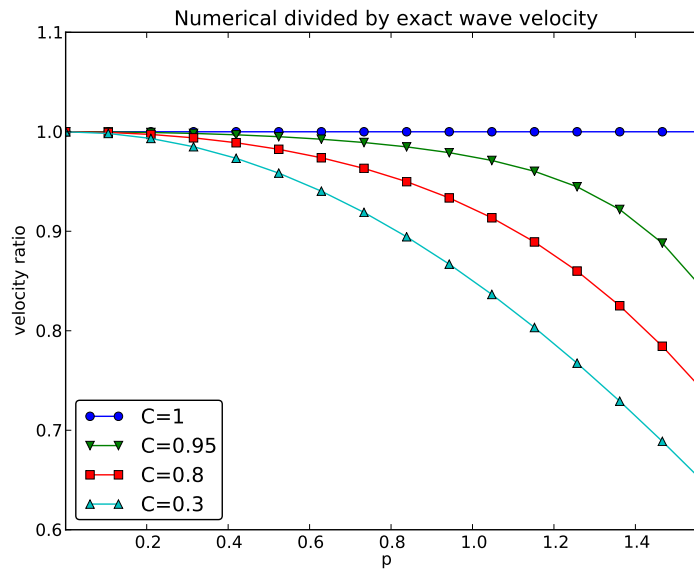
- Study error in wave velocity through  $\tilde{c}/c$  as function of  $p$

$$r(C, p) = \frac{\tilde{c}}{c} = \frac{2}{kc\Delta t} \sin^{-1}(C \sin p) = \frac{2}{kC\Delta x} \sin^{-1}(C \sin p) = \frac{1}{Cp} \sin^{-1}(C \sin p)$$

Can plot  $r(C, p)$  for  $p \in (0, \pi/2]$ ,  $C \in (0, 1]$

## Visualizing the error in wave velocity

```
def r(C, p):
    return 1/(C*p)*asin(C*sin(p))
```



Note: the shortest waves have the largest error, and short waves move too slowly.

## Taylor expanding the error in wave velocity

For small  $p$ , Taylor expand  $\tilde{\omega}$  as polynomial in  $p$ :

```
>> C, p = symbols('C p')
>> rs = r(C, p).series(p, 0, 7)
>> print rs
1 - p**2/6 + p**4/120 - p**6/5040 + C**2*p**2/6 -
C**2*p**4/12 + 13*C**2*p**6/720 + 3*C**4*p**4/40 -
C**4*p**6/16 + 5*C**6*p**6/112 + O(p**7)

>> # Drop the remainder O(...) term
>> rs = rs.removeO()
>> # Factorize each term
>> rs = [factor(term) for term in rs.as_ordered_terms()]
>> rs = sum(rs)
```



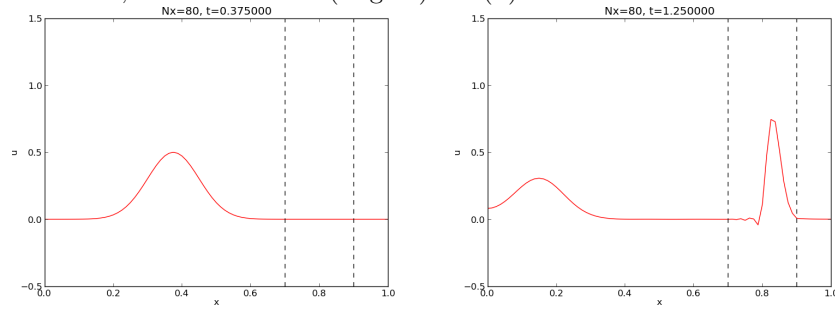
```
>> print rs
p**6*(C - 1)*(C + 1)*(225*C**4 - 90*C**2 + 1)/5040 +
p**4*(C - 1)*(C + 1)*(3*C - 1)*(3*C + 1)/120 +
p**2*(C - 1)*(C + 1)/6 + 1
```

Leading error term is  $\frac{1}{6}(C^2 - 1)p^2$  or

$$\frac{1}{6} \left( \frac{k\Delta x}{2} \right)^2 (C^2 - 1) = \frac{k^2}{24} (c^2 \Delta t^2 - \Delta x^2) = \mathcal{O}(\Delta t^2, \Delta x^2)$$

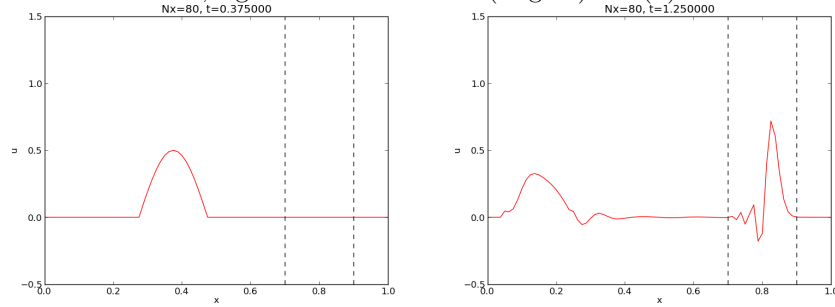
### Example on effect of wrong wave velocity (1)

Smooth wave, few short waves (large  $k$ ) in  $I(x)$ :



### Example on effect of wrong wave velocity (1)

Not so smooth wave, significant short waves (large  $k$ ) in  $I(x)$ :



### Stability

$$\sin \left( \frac{\tilde{\omega} \Delta t}{2} \right) = C \sin \left( \frac{k \Delta x}{2} \right)$$

- Exact  $\omega$  is real
- Complex  $\tilde{\omega}$  will lead to exponential growth of the amplitude
- Stability criterion: real  $\tilde{\omega}$

- Then  $\sin(\tilde{\omega}\Delta t/2) \in [-1, 1]$
- $k\Delta x/2$  is always real, so right-hand side is in  $[-C, C]$
- Then we must have  $C \leq 1$

Stability criterion:

$$C = \frac{c\Delta t}{\Delta x} \leq 1$$

### Why $C > 1$ leads to non-physical waves

Recall that right-hand side is in  $[-C, C]$ . Then  $C > 1$  means

$$\underbrace{\sin\left(\frac{\tilde{\omega}\Delta t}{2}\right)}_{>1} = C \sin\left(\frac{k\Delta x}{2}\right)$$

- $|\sin x| > 1$  implies complex  $x$
- Here: complex  $\tilde{\omega} = \tilde{\omega}_r \pm i\tilde{\omega}_i$
- One  $\tilde{\omega}_i < 0$  gives  $\exp(i \cdot i\tilde{\omega}_i) = \exp(-\tilde{\omega}_i)$  and *exponential growth*
- This wave component will after some time dominate the solution give an overall exponentially increasing amplitude (non-physical!)

### Extending the analysis to 2D (and 3D)

$$u(x, y, t) = g(k_x x + k_y y - \omega t)$$

is a typically solution of

$$u_{tt} = c^2(u_{xx} + u_{yy})$$

Can build solutions by adding complex Fourier components of the form

$$e^{i(k_x x + k_y y - \omega t)}$$

### Discrete wave components in 2D

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u)]_{q,r}^n$$

This equation admits a Fourier component

$$u_{q,r}^n = e^{i(k_x q \Delta x + k_y r \Delta y - \tilde{\omega} n \Delta t)}$$

Inserting the expression and using formulas from the 1D analysis:

$$\sin^2 \left( \frac{\tilde{\omega} \Delta t}{2} \right) = C_x^2 \sin^2 p_x + C_y^2 \sin^2 p_y$$

where

$$C_x = \frac{c^2 \Delta t^2}{\Delta x^2}, \quad C_y = \frac{c^2 \Delta t^2}{\Delta y^2}, \quad p_x = \frac{k_x \Delta x}{2}, \quad p_y = \frac{k_y \Delta y}{2}$$

### Stability criterion in 2D

Real-valued  $\tilde{\omega}$  requires

$$C_x^2 + C_y^2 \leq 1$$

or

$$\Delta t \leq \frac{1}{c} \left( \frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} \right)^{-1/2}$$

### Stability criterion in 3D

$$\Delta t \leq \frac{1}{c} \left( \frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2} \right)^{-1/2}$$

For  $c^2 = c^2(\mathbf{x})$  we must use the worst-case value  $\bar{c} = \sqrt{\max_{\mathbf{x} \in \Omega} c^2(\mathbf{x})}$  and a safety factor  $\beta \leq 1$ :

$$\Delta t \leq \beta \frac{1}{\bar{c}} \left( \frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2} \right)^{-1/2}$$

### Numerical dispersion relation in 2D (1)

$$\tilde{\omega} = \frac{2}{\Delta t} \sin^{-1} \left( \left( C_x^2 \sin^2 p_x + C_y^2 \sin^2 p_y \right)^{\frac{1}{2}} \right)$$

For visualization, introduce  $\theta$ :

$$k_x = k \sin \theta, \quad k_y = k \cos \theta, \quad p_x = \frac{1}{2} k h \cos \theta, \quad p_y = \frac{1}{2} k h \sin \theta$$

Also:  $\Delta x = \Delta y = h$ . Then  $C_x = C_y = c \Delta t / h \equiv C$ .

Now  $\tilde{\omega}$  depends on

- $C$  reflecting the number cells a wave is displaced during a time step
- $kh$  reflecting the number of cells per wave length in space
- $\theta$  expressing the direction of the wave

### Numerical dispersion relation in 2D (2)

$$\frac{\tilde{c}}{c} = \frac{1}{Ckh} \sin^{-1} \left( C \left( \sin^2\left(\frac{1}{2}kh \cos \theta\right) + \sin^2\left(\frac{1}{2}kh \sin \theta\right) \right)^{\frac{1}{2}} \right)$$

Can make color contour plots of  $1 - \tilde{c}/c$  in *polar coordinates* with  $\theta$  as the angular coordinate and  $kh$  as the radial coordinate.

### Numerical dispersion relation in 2D (3)

